Lecture 12: Feature importances

UBC 2022-23

Instructor: Mathias Lécuyer

Imports

In [1]:
```
import os
import string
import sys
from collections import deque

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

sys.path.append("../code/.")

import seaborn as sns
from plotting_functions import *
from sklearn import datasets
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier, DummyRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegres
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, Standa
from sklearn.svm import SVC, SVR
from sklearn.tree import DecisionTreeClassifier
from utils import *

%matplotlib inline
```

Learning outcomes

From this lecture, students are expected to be able to:

- Interpret the coefficients of linear regression for ordinal, one-hot encoded categorical, and scaled numeric features.

- Explain why interpretability is important in ML.
- Use `feature_importances_` attribute of `sklearn` models and interpret its output.
- Use `eli5` to get feature importances of non `sklearn` models and interpret its output.
- Apply SHAP to assess feature importances and interpret model predictions.
- Explain force plot, summary plot, and dependence plot produced with shapely values.

In [2]:
```
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
```

Data

In this lecture, we'll be using [Kaggle House Prices dataset (https://www.kaggle.com/c/home-data-for-ml-course/)](https://www.kaggle.com/c/home-data-for-ml-course/), the dataset we used in lecture 2. As usual, to run this notebook you'll need to download the data. Unzip the data into a subdirectory called `data`. For this dataset, train and test have already been separated. We'll be working with the train portion in this lecture.

In [3]:
```
df = pd.read_csv("../data/housing-kaggle/train.csv")
train_df, test_df = train_test_split(df, test_size=0.10, random_state=1
train_df.head()
```

Out[3]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	U
302	303	20	RL	118.0	13704	Pave	NaN	IR1	Lvl	
767	768	50	RL	75.0	12508	Pave	NaN	IR1	Lvl	
429	430	20	RL	130.0	11457	Pave	NaN	IR1	Lvl	
1139	1140	30	RL	98.0	8731	Pave	NaN	IR1	Lvl	
558	559	60	RL	57.0	21872	Pave	NaN	IR2	HLS	

5 rows × 81 columns

- The prediction task is predicting `SalePrice` given features related to properties.
- Note that the target is numeric, not categorical (it's a regression problem).

In [4]:
```
train_df.shape
```

Out[4]: (1314, 81)

Let's separate `X` and `y`

In [5]:
```
X_train = train_df.drop(columns=["SalePrice"])
y_train = train_df["SalePrice"]

X_test = test_df.drop(columns=["SalePrice"])
y_test = test_df["SalePrice"]
```

Let's identify feature types

In [6]:
```
drop_features = ["Id"]
numeric_features = [
    "BedroomAbvGr",
    "KitchenAbvGr",
    "LotFrontage",
    "LotArea",
    "OverallQual",
    "OverallCond",
    "YearBuilt",
    "YearRemodAdd",
    "MasVnrArea",
    "BsmtFinSF1",
    "BsmtFinSF2",
    "BsmtUnfSF",
    "TotalBsmtSF",
    "1stFlrSF",
    "2ndFlrSF",
    "LowQualFinSF",
    "GrLivArea",
    "BsmtFullBath",
    "BsmtHalfBath",
    "FullBath",
    "HalfBath",
    "TotRmsAbvGrd",
    "Fireplaces",
    "GarageYrBlt",
    "GarageCars",
    "GarageArea",
    "WoodDeckSF",
    "OpenPorchSF",
    "EnclosedPorch",
    "3SsnPorch",
    "ScreenPorch",
    "PoolArea",
    "MiscVal",
    "YrSold",
]
```

In [7]:
```
ordinal_features_reg = [
    "ExterQual",
    "ExterCond",
    "BsmtQual",
    "BsmtCond",
    "HeatingQC",
    "KitchenQual",
    "FireplaceQu",
    "GarageQual",
    "GarageCond",
    "PoolQC",
]
ordering = [
    "Po",
    "Fa",
    "TA",
    "Gd",
    "Ex",
]  # if N/A it will just impute something, per below
ordering_ordinal_reg = [ordering] * len(ordinal_features_reg)
ordering_ordinal_reg
```

Out[7]:
```
[['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex'],
 ['Po', 'Fa', 'TA', 'Gd', 'Ex']]
```

In [8]:
```
ordinal_features_oth = [
    "BsmtExposure",
    "BsmtFinType1",
    "BsmtFinType2",
    "Functional",
    "Fence",
]
ordering_ordinal_oth = [
    ["NA", "No", "Mn", "Av", "Gd"],
    ["NA", "Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    ["NA", "Unf", "LwQ", "Rec", "BLQ", "ALQ", "GLQ"],
    ["Sal", "Sev", "Maj2", "Maj1", "Mod", "Min2", "Min1", "Typ"],
    ["NA", "MnWw", "GdWo", "MnPrv", "GdPrv"],
]
```

In [9]:
```
categorical_features = list(
    set(X_train.columns)
    - set(numeric_features)
    - set(ordinal_features_reg)
    - set(ordinal_features_oth)
    - set(drop_features)
)
categorical_features
```

Out[9]:
```
['Electrical',
 'LotShape',
 'Exterior1st',
 'MiscFeature',
 'LandContour',
 'RoofMatl',
 'Foundation',
 'MSZoning',
 'LandSlope',
 'SaleType',
 'Street',
 'HouseStyle',
 'Condition1',
 'GarageFinish',
 'Heating',
 'Neighborhood',
 'Exterior2nd',
 'Condition2',
 'PavedDrive',
 'MSSubClass',
 'Utilities',
 'Alley',
 'MasVnrType',
 'RoofStyle',
 'SaleCondition',
 'BldgType',
 'CentralAir',
 'MoSold',
 'LotConfig',
 'GarageType']
```

In [10]:

```
from sklearn.compose import ColumnTransformer, make_column_transformer

numeric_transformer = make_pipeline(SimpleImputer(strategy="median"), S
ordinal_transformer_reg = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_reg),
)

ordinal_transformer_oth = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OrdinalEncoder(categories=ordering_ordinal_oth),
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse=False),
)

preprocessor = make_column_transformer(
    ("drop", drop_features),
    (numeric_transformer, numeric_features),
    (ordinal_transformer_reg, ordinal_features_reg),
    (ordinal_transformer_oth, ordinal_features_oth),
    (categorical_transformer, categorical_features),
)
```

In [11]:
```
preprocessor.fit(X_train)
preprocessor.named_transformers_
```

Out[11]:
```
{'drop': 'drop',
 'pipeline-1': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='median')),
                 ('standardscaler', StandardScaler())]),
 'pipeline-2': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='most_frequent')),
                 ('ordinalencoder',
                  OrdinalEncoder(categories=[['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x'],
                                             ['Po', 'Fa', 'TA', 'Gd', 'E
x']]))]),
 'pipeline-3': Pipeline(steps=[('simpleimputer', SimpleImputer(strategy
='most_frequent')),
                 ('ordinalencoder',
                  OrdinalEncoder(categories=[['NA', 'No', 'Mn', 'Av', 'G
d'],
                                             ['NA', 'Unf', 'LwQ', 'Rec',
'BLQ',
                                              'ALQ', 'GLQ'],
                                             ['NA', 'Unf', 'LwQ', 'Rec',
'BLQ',
                                              'ALQ', 'GLQ'],
                                             ['Sal', 'Sev', 'Maj2', 'Maj
1',
                                              'Mod', 'Min2', 'Min1', 'Ty
p'],
                                             ['NA', 'MnWw', 'GdWo', 'MnPr
v',
                                              'GdPrv']]))]),
 'pipeline-4': Pipeline(steps=[('simpleimputer',
                  SimpleImputer(fill_value='missing', strategy='constan
t')),
                 ('onehotencoder',
                  OneHotEncoder(handle_unknown='ignore', sparse=Fals
e))])}
```

In [12]:
```
ohe_columns = list(
    preprocessor.named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names(categorical_features)
)
new_columns = (
    numeric_features + ordinal_features_reg + ordinal_features_oth + oh
)
```

In [13]:
```
X_train_enc = pd.DataFrame(
    preprocessor.transform(X_train), index=X_train.index, columns=new_c
)
X_train_enc
```

Out[13]:

	BedroomAbvGr	KitchenAbvGr	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	Y
302	0.154795	-0.222647	2.312501	0.381428	0.663680	-0.512408	0.993969	
767	1.372763	-0.222647	0.260890	0.248457	-0.054669	1.285467	-1.026793	
429	0.154795	-0.222647	2.885044	0.131607	-0.054669	-0.512408	0.563314	
1139	0.154795	-0.222647	1.358264	-0.171468	-0.773017	-0.512408	-1.689338	
558	0.154795	-0.222647	-0.597924	1.289541	0.663680	-0.512408	0.828332	
...	...	...	...	...	...	...	...	
1041	1.372763	-0.222647	-0.025381	-0.127107	-0.054669	2.184405	-0.165485	
1122	0.154795	-0.222647	-0.025381	-0.149788	-1.491366	-2.310284	-0.496757	
1346	0.154795	-0.222647	-0.025381	1.168244	0.663680	1.285467	-0.099230	
1406	-1.063173	-0.222647	0.022331	-0.203265	-0.773017	1.285467	0.033279	
1389	0.154795	-0.222647	-0.454788	-0.475099	-0.054669	0.386530	-0.993666	

1314 rows × 263 columns

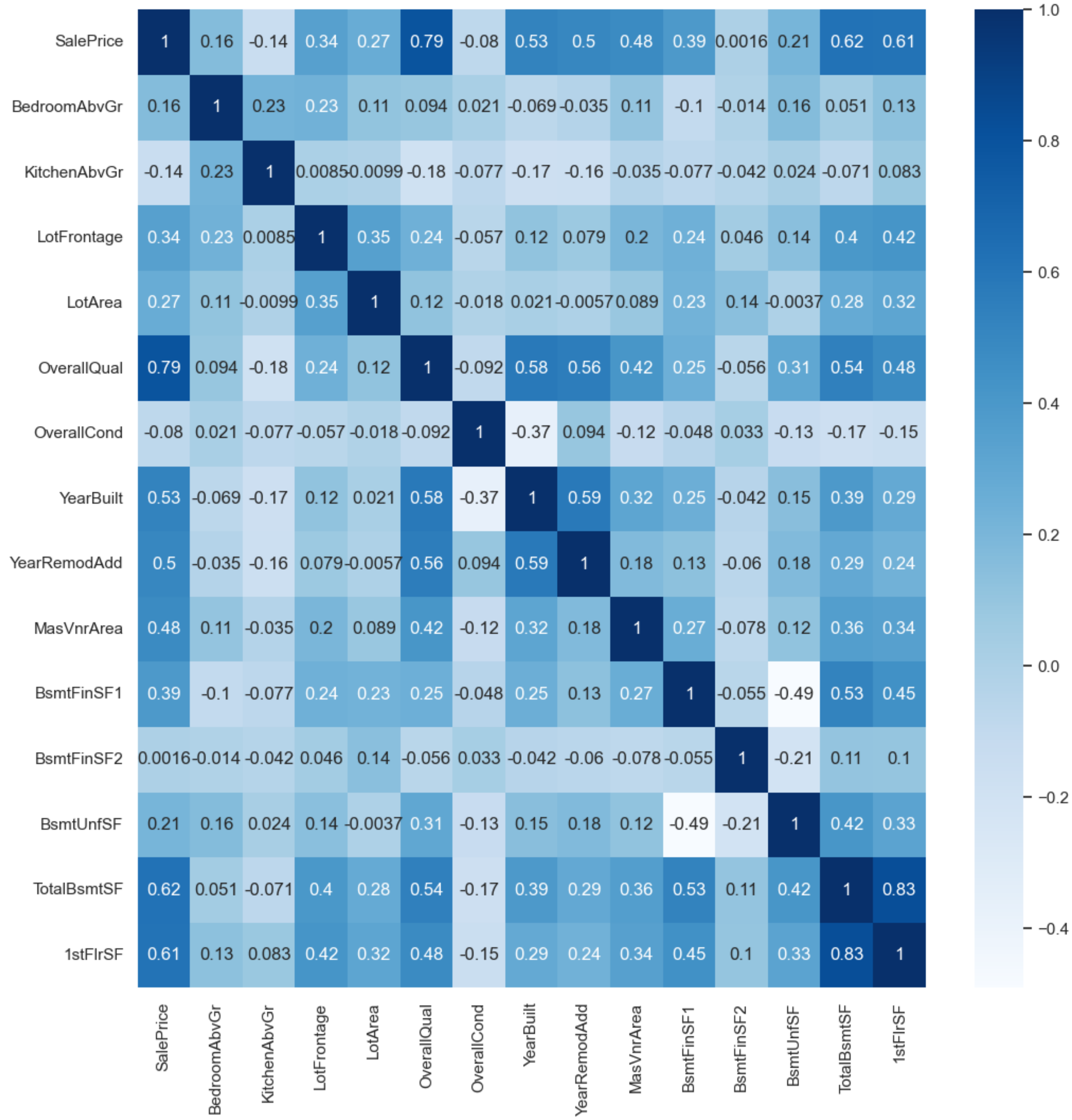
In [14]:
```
X_train_enc.shape
```

Out[14]: (1314, 263)

Feature correlations

- Let's look at the correlations between various features with other features and the target in our encoded data.
- In simple terms here is how you can interpret correlations between two variables X and Y:
    - If Y goes up when X goes up, we say X and Y are positively correlated.
    - If Y goes down when X goes up, we say X and Y are negatively correlated.
    - If Y does not change in predictable ways when X changes, we say X and Y are uncorrelated.

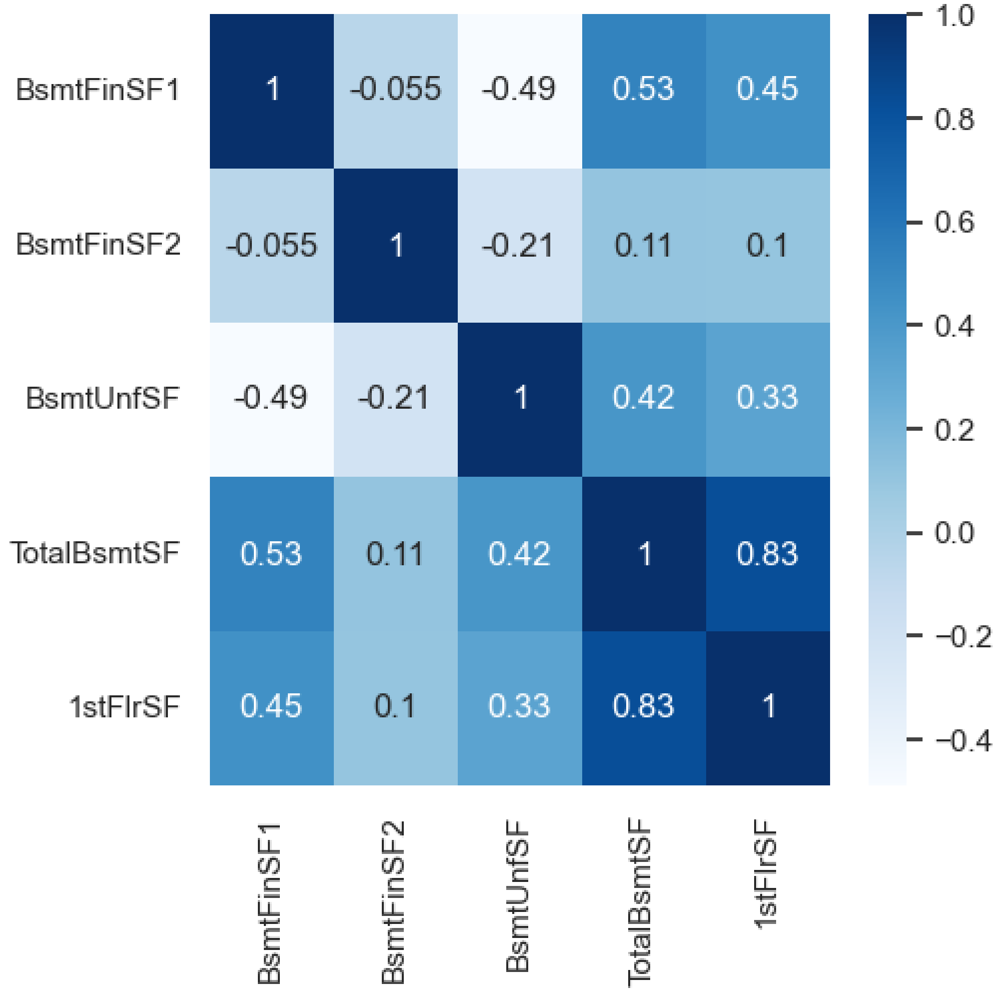
In [15]:
```
cor = pd.concat((y_train, X_train_enc), axis=1).iloc[:, :15].corr()
plt.figure(figsize=(12, 12))
sns.set(font_scale=1)
sns.heatmap(cor, annot=True, cmap=plt.cm.Blues);
```



- We can immediately see that `SalePrice` is highly correlated with `OverallQual`.
- This is an early hint that `OverallQual` is a useful feature in predicting `SalePrice`.
- However, this approach is **extremely simplistic**.
  - It only looks at each feature in isolation.
  - It only looks at linear associations:
    - What if `SalePrice` is high when `BsmtFullBath` is 2 or 3, but low when it's 0, 1, or 4? They might seem uncorrelated.

In [16]:

```
cor = pd.concat((y_train, X_train_enc), axis=1).iloc[:, 10:15].corr()
plt.figure(figsize=(5, 5))
sns.set(font_scale=1)
sns.heatmap(cor, annot=True, cmap=plt.cm.Blues);
```



- Looking at this diagram also tells us the relationship between features.
  - For example, `1stFlrSF` and `TotalBsmtSF` are highly correlated.
  - Do we need both of them?
  - If our model says `1stFlrSF` is very important and `TotalBsmtSF` is very unimportant, do we trust those values?
  - Maybe `TotalBsmtSF` only "becomes important" if `1stFlrSF` is removed.
  - Sometimes the opposite happens: a feature only becomes important if another feature is *added*.

Feature importances in linear models

- Like logistic regression, with linear regression we can look at the *coefficients* for each feature.
- Overall idea: predicted price = intercept + $\sum_i$ coefficient i $\times$ feature i.

In [17]:
```
lr = make_pipeline(preprocessor, Ridge())
lr.fit(X_train, y_train);
```

Let's look at the coefficients.

In [18]:
```
lr_coefs = pd.DataFrame(data=lr[1].coef_, index=new_columns, columns=["
lr_coefs.head(20)
```

Out[18]:

	Coefficient
BedroomAbvGr	-3723.741570
KitchenAbvGr	-4580.204576
LotFrontage	-1578.664421
LotArea	5109.356718
OverallQual	12487.561839
OverallCond	4855.535334
YearBuilt	4226.684842
YearRemodAdd	324.664715
MasVnrArea	5251.325210
BsmtFinSF1	3667.172851
BsmtFinSF2	583.114880
BsmtUnfSF	-1266.614671
TotalBsmtSF	2751.084018
1stFlrSF	6736.788904
2ndFlrSF	13409.901084
LowQualFinSF	-448.424132
GrLivArea	15988.182407
BsmtFullBath	2299.227266
BsmtHalfBath	500.169112
FullBath	2831.811467

Interpreting coefficients of different types of features.

Ordinal features

- The ordinal features are easiest to interpret.

In [19]:
```
print(ordinal_features_reg)
```

```
['ExterQual', 'ExterCond', 'BsmtQual', 'BsmtCond', 'HeatingQC', 'KitchenQ
ual', 'FireplaceQu', 'GarageQual', 'GarageCond', 'PoolQC']
```

In [20]:
```
lr_coefs.loc["ExterQual", "Coefficient"]
```

Out[20]:
```
4195.671512467474
```

- Increasing by one category of exterior quality (e.g. good -> excellent) increases the predicted price by $\sim$ $4195.
  - Wow, that's a lot!
  - Remember this is just what the model has learned. It doesn't tell us how the world works.

In [21]:
```
one_example = X_test[:1]
```

In [22]:
```
one_example["ExterQual"]
```

Out[22]:
```
147    Gd
Name: ExterQual, dtype: object
```

Let's perturb the example and change `ExterQual` to `Ex`.

In [23]:
```
one_example_perturbed = one_example.copy()
one_example_perturbed["ExterQual"] = "Ex"  # Change Gd to Ex
```

In [24]:
```
one_example_perturbed
```

Out[24]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Util
147	148	60	RL	NaN	9505	Pave	NaN	IR1	Lvl	Al

1 rows × 80 columns

In [25]:
```
one_example_perturbed["ExterQual"]
```

Out[25]:
```
147    Ex
Name: ExterQual, dtype: object
```

How does the prediction change after changing `ExterQual` from `Gd` to `Ex`?

In [26]:
```
print("Prediction on the original example: ", lr.predict(one_example))
print("Prediction on the perturbed example: ", lr.predict(one_example_p
print(
    "After changing ExterQual from Gd to Ex increased the prediction by
    lr.predict(one_example_perturbed) - lr.predict(one_example),
)
```

```
Prediction on the original example:  [224795.63596803]
Prediction on the perturbed example:  [228991.30748049]
After changing ExterQual from Gd to Ex increased the prediction by:  [419
5.67151247]
```

That's exactly the learned coefficient for `ExterQual` !

In [27]:
```
lr_coefs.loc["ExterQual", "Coefficient"]
```

Out[27]:
```
4195.671512467474
```

So our interpretation is correct!

- Increasing by one category of exterior quality (e.g. good -> excellent) increases the predicted price by $\sim$ $4195.

Categorical features

- What about the categorical features?
- We have created a number of columns for each category with OHE and each category gets it's own coefficient.

In [28]:
```
print(categorical_features)
```

```
['Electrical', 'LotShape', 'Exterior1st', 'MiscFeature', 'LandContour',
'RoofMatl', 'Foundation', 'MSZoning', 'LandSlope', 'SaleType', 'Street',
'HouseStyle', 'Condition1', 'GarageFinish', 'Heating', 'Neighborhood', 'E
xterior2nd', 'Condition2', 'PavedDrive', 'MSSubClass', 'Utilities', 'Alle
y', 'MasVnrType', 'RoofStyle', 'SaleCondition', 'BldgType', 'CentralAir',
'MoSold', 'LotConfig', 'GarageType']
```

In [29]:
```
lr_coefs_landslope = lr_coefs[lr_coefs.index.str.startswith("LandSlope"
lr_coefs_landslope
```

Out[29]:

	Coefficient
LandSlope_Gtl	457.197456
LandSlope_Mod	7420.208381
LandSlope_Sev	-7877.405837

- We can talk about switching from one of these categories to another by picking a "reference" category:

In [30]:
```
lr_coefs_landslope - lr_coefs_landslope.loc["LandSlope_Gtl"]
```

Out[30]:

	Coefficient
LandSlope_Gtl	0.000000
LandSlope_Mod	6963.010925
LandSlope_Sev	-8334.603292

- If you change the category from `LandSlope_Gtl` to `LandSlope_Mod` the prediction price goes up by $\sim \$6963$
- If you change the category from `LandSlope_Gtl` to `LandSlope_Sev` the prediction price goes down by $\sim \$8334$

Note that this might not make sense in the real world but this is what our model decided to learn given this small amount of data.

In [31]:
```
lr_coefs.sort_values(by="Coefficient")
```

Out[31]:

	Coefficient
RoofMatl_ClyTile	-191129.774314
Condition2_PosN	-105552.840565
Heating_OthW	-27260.681308
MSZoning_C (all)	-21990.746193
Exterior1st_ImStucc	-19393.964621
...	...
PoolQC	34217.656047
RoofMatl_CompShg	36525.980874
Neighborhood_NridgHt	37532.643270
Neighborhood_StoneBr	39993.978324
RoofMatl_WdShngl	83646.711008

263 rows × 1 columns

- For example, the above coefficient says that "If the roof is made of clay or tile, the predicted price is $191K less"?
- Do we believe these interpretations??
  - Do we believe this is how the predictions are being computed? Yes.
  - Do we believe that this is how the world works? No.

If you did `drop='first'` (we didn't) then you already have a reference class, and all the values are with respect to that one. **The interpretation depends on the variable encoding**, here whether we did `drop='first'`.

Interpreting coefficients of numeric features

Let's look at coefficients of `PoolArea` and `LotFrontage`.

In [32]:
```
lr_coefs.loc[["PoolArea", "LotArea"]]
```

Out[32]:

	Coefficient
PoolArea	2822.370476
LotArea	5109.356718

Intuition:

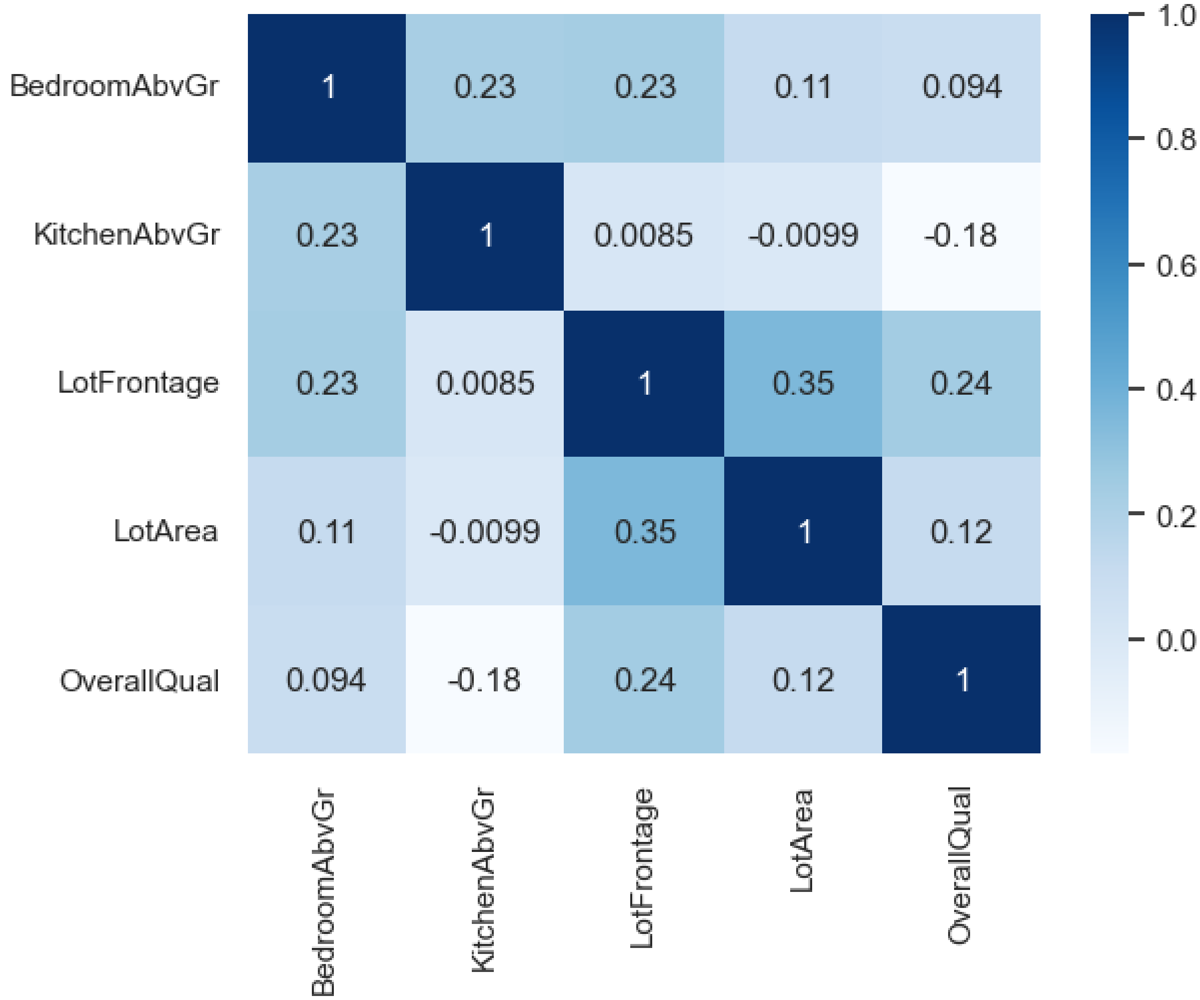
- Tricky because numeric features are **scaled**!
- **Increasing** `PoolArea` by 1 scaled unit **increases** the predicted price by $\sim \$2822$.
- **Increasing** `LotFrontage` by 1 scaled unit **decreases** the predicted price by $\sim \$1578$.

Does that sound reasonable?

- For `PoolArea`, yes.
- For `LotFrontage`, that's surprising. Something positive would have made more sense?

It's not the case here but maybe the problem is that `LotFrontage` and `LotArea` are very correlated. `LotArea` has a larger positive coefficient.

In [33]:
```
cor = X_train_enc[numeric_features[:5]].corr()
sns.heatmap(cor, annot=True, cmap=plt.cm.Blues);
```



BTW, let's make sure the predictions behave as expected:

In [34]:
```
one_example = X_test[:1]
```

In [35]:
```
one_example
```

Out[35]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Util
147	148	60	RL	NaN	9505	Pave	NaN	IR1	Lvl	Al

1 rows × 80 columns

Let's perturb the example and add 1 to the `LotArea` .

In [36]:
```
one_example_perturbed = one_example.copy()
one_example_perturbed["LotArea"] += 1  # add 1 to the LotArea
```

In [37]:
```
one_example_perturbed
```

Out[37]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Util
147	148	60	RL	NaN	9506	Pave	NaN	IR1	Lvl	Al

1 rows × 80 columns

Prediction on the original example.

In [38]:
```
lr.predict(one_example)
```

Out[38]: array([224795.63596803])

Prediction on the perturbed example.

In [39]:
```
lr.predict(one_example_perturbed)
```

Out[39]: array([224796.2040233])

- What's the difference between prediction?
- Does the difference make sense given the coefficient of the feature?

In [40]:
```
lr.predict(one_example_perturbed) - lr.predict(one_example)
```

Out[40]: array([0.56805528])

In [41]:
```
lr_coefs.loc[["LotArea"]]
```

Out[41]:

	Coefficient
LotArea	5109.356718

- Why did the prediction only go up by \$0.57 instead of \$5109?
- The scaling happens **after our change**, so this is an issue of units: `LotArea` is in sqft, but the coefficient is not $\$5109/\text{sqft}$ **because we scaled the features**.

Example showing how to interpret coefficients of scaled features.

- The scaler subtracted the mean and divided by the standard deviation.
- The division actually changed the scale!
- For the unit conversion, we don't care about the subtraction, but only the scaling.

In [42]:
```
scaler = preprocessor.named_transformers_["pipeline-1"]["standardscaler
```

In [43]:
```
np.sqrt(scaler.var_)
```

Out[43]:
```
array([8.21039683e-01, 2.18760172e-01, 2.09591390e+01, 8.99447103e+03,
       1.39208177e+00, 1.11242416e+00, 3.01866337e+01, 2.06318985e+01,
       1.77914527e+02, 4.59101890e+02, 1.63890010e+02, 4.42869860e+02,
       4.42817167e+02, 3.92172897e+02, 4.35820743e+02, 4.69800920e+01,
       5.29468070e+02, 5.18276015e-01, 2.33809970e-01, 5.49298599e-01,
       5.02279069e-01, 1.62604030e+00, 6.34398801e-01, 2.40531598e+01,
       7.40269201e-01, 2.10560601e+02, 1.25388753e+02, 6.57325181e+01,
       6.07432962e+01, 3.03088902e+01, 5.38336322e+01, 4.23249944e+01,
       5.22084645e+02, 1.33231649e+00])
```

In [44]:
```
lr_scales = pd.DataFrame(
    data=np.sqrt(scaler.var_), index=numeric_features, columns=["Scale"
)
lr_scales.head()
```

Out[44]:

	Scale
BedroomAbvGr	0.821040
KitchenAbvGr	0.218760
LotFrontage	20.959139
LotArea	8994.471032
OverallQual	1.392082

- It seems like `LotArea` was divided by 8994.471032 sqft.

In [45]:
```
lr_coefs.loc["LotArea", "Coefficient"]
```

Out[45]:
```
5109.356718094066
```

In [46]:
```
lr_coefs.loc["LotArea", "Coefficient"] / lr_scales.loc["LotArea", "Scal
```

Out[46]:
```
0.5680552752646618
```

In [47]:
```
lr_coefs.loc[["LotArea"]]
```

Out[47]:

	Coefficient
LotArea	5109.356718

- The coefficient tells us that if we increase the **scaled** `LotArea` by one unit the price would go up by $\approx \$5109$.
- One scaled unit represents ~ 8994 sq feet.

- So if I increase original `LotArea` by one square foot then the predicted price would go up by this amount:

In [48]:
```
5109.356718094072 / 8994.471032
```
Out[48]:
```
0.5680552752814816
```

- This makes much more sense. Now we get the number we got before.
- That said don't read too much into these coefficients without statistical training.

Interim summary

- Correlation among features might make coefficients completely uninterpretable.
- Fairly straightforward to interpret coefficients of ordinal features.
- In categorical features, it's often helpful to consider one category as a reference point and think about relative importance.
- For numeric features, relative importance is meaningful after scaling.
- You have to be careful about the scale of the feature when interpreting the coefficients.
- Remember that explaining the model $\neq$ explaining the data.
- the coefficients tell us only about the model and they might not accurately reflect the data.

Break (5 min)

Interpretability of ML models: Motivations

Why model interpretability?

- Ability to interpret ML models is crucial in many applications such as banking, healthcare, and criminal justice.
- It can be leveraged by domain experts to diagnose systematic errors and underlying biases of complex ML systems.

[Source (https://github.com/slundberg/shap/blob/master/docs/presentations/February%202018%20Talk.ppt](https://github.com/slundberg/shap/blob/master/docs/presentations/February%202018%20Talk.ppt)

What is model interpretability?

- In this course, our definition of model iterpretability will be looking at **feature importances.**
- There is more to interpretability than feature importances, but it's a good start!

- Resource:
  - [Interpretable Machine Learning (https://christophm.github.io/interpretable-ml-book/interpretability-importance.html)](https://christophm.github.io/interpretable-ml-book/interpretability-importance.html)
  - [Yann LeCun, Kilian Weinberger, Patrice Simard, and Rich Caruana: Panel debate on interpretability (https://vimeo.com/252187813)](https://vimeo.com/252187813)

Data

- Let's work with [the adult census data set (https://www.kaggle.com/uciml/adult-census-income)](https://www.kaggle.com/uciml/adult-census-income) from last lecture.

In [49]:
```python
adult_df_large = pd.read_csv("../data/adult.csv")
train_df, test_df = train_test_split(adult_df_large, test_size=0.2, ran
train_df_nan = train_df.replace("?", np.NaN)
test_df_nan = test_df.replace("?", np.NaN)
train_df_nan.head()
```

Out[49]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	
5514	26	Private	256263	HS-grad	9	Never-married	Craft-repair	Not-in-family	\
19777	24	Private	170277	HS-grad	9	Never-married	Other-service	Not-in-family	\
10781	36	Private	75826	Bachelors	13	Divorced	Adm-clerical	Unmarried	\
32240	22	State-gov	24395	Some-college	10	Married-civ-spouse	Adm-clerical	Wife	\
9876	31	Local-gov	356689	Bachelors	13	Married-civ-spouse	Prof-specialty	Husband	\

In [50]:
```python
numeric_features = ["age", "fnlwgt", "capital.gain", "capital.loss", "h
categorical_features = [
    "workclass",
    "marital.status",
    "occupation",
    "relationship",
    "native.country",
]
ordinal_features = ["education"]
binary_features = ["sex"]
drop_features = ["race", "education.num"]
target_column = "income"
```

In [51]:
```
education_levels = [
    "Preschool",
    "1st-4th",
    "5th-6th",
    "7th-8th",
    "9th",
    "10th",
    "11th",
    "12th",
    "HS-grad",
    "Prof-school",
    "Assoc-voc",
    "Assoc-acdm",
    "Some-college",
    "Bachelors",
    "Masters",
    "Doctorate",
]
```

In [52]:
```
assert set(education_levels) == set(train_df["education"].unique())
```

In [53]:
```
numeric_transformer = make_pipeline(SimpleImputer(strategy="median"), S
tree_numeric_transformer = make_pipeline(SimpleImputer(strategy="median

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore"),
)

ordinal_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OrdinalEncoder(categories=[education_levels], dtype=int),
)

binary_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(drop="if_binary", dtype=int),
)

preprocessor = make_column_transformer(
    ("drop", drop_features),
    (numeric_transformer, numeric_features),
    (ordinal_transformer, ordinal_features),
    (binary_transformer, binary_features),
    (categorical_transformer, categorical_features),
)
```

In [54]:
```
X_train = train_df_nan.drop(columns=[target_column])
y_train = train_df_nan[target_column]

X_test = test_df_nan.drop(columns=[target_column])
y_test = test_df_nan[target_column]
```

Do we have class imbalance?

- There is class imbalance. But without any context, both classes seem equally important.
- Let's use accuracy as our metric.

In [55]:
```
train_df_nan["income"].value_counts(normalize=True)
```

Out[55]:
```
<=50K    0.757985
>50K     0.242015
Name: income, dtype: float64
```

In [56]:
```
scoring_metric = "accuracy"
```

Let's store all the results in a dictionary called `results` .

In [57]:
```
results = {}
```

In [58]:
```
scoring_metric = "accuracy"
```

In [59]:
```
from lightgbm.sklearn import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder

pipe_lr = make_pipeline(
    preprocessor, LogisticRegression(max_iter=2000, random_state=123)
)
pipe_rf = make_pipeline(preprocessor, RandomForestClassifier(random_sta
pipe_xgb = make_pipeline(
    preprocessor, XGBClassifier(random_state=123, eval_metric="logloss"
)
pipe_lgbm = make_pipeline(preprocessor, LGBMClassifier(random_state=123

# XGBoost requires numeric targets
label_encoder = LabelEncoder()
label_encoder.fit(y_train)

classifiers = {
    "logistic regression": pipe_lr,
    "random forest": pipe_rf,
    "XGBoost": pipe_xgb,
    "LightGBM": pipe_lgbm,
}
```

In [60]:
```
dummy = DummyClassifier(strategy="stratified")
results["Dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True, scoring=scoring_m
)
```

In [61]:
```
for (name, model) in classifiers.items():
    results[name] = mean_std_cross_val_scores(
        model, X_train, label_encoder.transform(y_train), return_train_
    )
```

In [62]:
```
pd.DataFrame(results).T
```

Out[62]:

	fit_time	score_time	test_score	train_score
Dummy	0.008 (+/- 0.001)	0.005 (+/- 0.000)	0.631 (+/- 0.002)	0.633 (+/- 0.002)
logistic regression	0.633 (+/- 0.056)	0.010 (+/- 0.000)	0.850 (+/- 0.006)	0.851 (+/- 0.001)
random forest	6.678 (+/- 0.022)	0.074 (+/- 0.001)	0.857 (+/- 0.004)	1.000 (+/- 0.000)
XGBoost	0.917 (+/- 0.126)	0.017 (+/- 0.001)	0.871 (+/- 0.004)	0.908 (+/- 0.001)
LightGBM	0.771 (+/- 0.086)	0.018 (+/- 0.001)	0.871 (+/- 0.004)	0.892 (+/- 0.000)

- One problem is that often simple models are interpretable but not accurate.
- But more complex models (e.g., LightGBM) are less interpretable.

Source
(https://github.com/slundberg/shap/blob/master/docs/presentations/February%202018%20Talk.ppt

Feature importances in linear models

- Simpler models are often more interpretable but less accurate.

Let's create and fit a pipeline with preprocessor and logistic regression.

In [63]:
```
pipe_lr = make_pipeline(preprocessor, LogisticRegression(max_iter=2000,
pipe_lr.fit(X_train, y_train);
```

In [64]:
```
ohe_feature_names = (
    pipe_rf.named_steps["columntransformer"]
    .named_transformers_["pipeline-4"]
    .named_steps["onehotencoder"]
    .get_feature_names()
    .tolist()
)
feature_names = (
    numeric_features + ordinal_features + binary_features + ohe_feature
)
feature_names[:10]
```

Out[64]:
```
['age',
 'fnlwgt',
 'capital.gain',
 'capital.loss',
 'hours.per.week',
 'education',
 'sex',
 'x0_Federal-gov',
 'x0_Local-gov',
 'x0_Never-worked']
```

In [65]:
```
data = {
    "coefficient": pipe_lr.named_steps["logisticregression"].coef_[0].t
    "magnitude": np.absolute(
        pipe_lr.named_steps["logisticregression"].coef_[0].tolist()
    ),
}
coef_df = pd.DataFrame(data, index=feature_names).sort_values(
    "magnitude", ascending=False
)
```

In [66]:
```
coef_df[:10]
```

Out[66]:

	coefficient	magnitude
capital.gain	2.355403	2.355403
x1_Married-AF-spouse	1.729264	1.729264
x2_Priv-house-serv	-1.408362	1.408362
x1_Married-civ-spouse	1.324596	1.324596
x3_Wife	1.261245	1.261245
x4_Columbia	-1.104853	1.104853
x2_Prof-specialty	1.064275	1.064275
x2_Exec-managerial	1.041384	1.041384
x3_Own-child	-1.014115	1.014115
x4_Dominican-Republic	-1.007272	1.007272

- Increasing `capital.gain` is likely to push the prediction towards ">50k" income class

- Whereas occupation of private house service is likely to push the prediction towards "<=50K" income.

Can we get feature importances for non-linear models?

Model interpretability beyond linear models

We will be looking at three ways for model interpretability.

- `sklearn` `feature_importances_`
- [eli5 (https://eli5.readthedocs.io/en/latest/tutorials/black-box-text-classifiers.html#lime-tutorial)](https://eli5.readthedocs.io/en/latest/tutorials/black-box-text-classifiers.html#lime-tutorial) (stands for "explain like I'm 5")
- [SHAP (https://github.com/slundberg/shap)](https://github.com/slundberg/shap)

`sklearn` `feature_importances_`

- Many `sklearn` models have `feature_importances_` attribute.
- For tree-based models it's calculated based on impurity (gini index or information gain).
- For example, let's look at `feature_importances_` of `RandomForestClassifier`.

Let's create and fit a pipeline with preprocessor and random forest.

In [67]:
```
pipe_rf = make_pipeline(preprocessor, RandomForestClassifier(random_sta
pipe_rf.fit(X_train, y_train);
```

Which features are driving the predictions the most?

In [68]:

```
data = {
    "Importance": pipe_rf.named_steps["randomforestclassifier"].feature
}
imps = pd.DataFrame(data=data, index=feature_names,).sort_values(
    by="Importance", ascending=False
)[:10]
imps
```

Out[68]:

	Importance
fnlwgt	0.169580
age	0.153339
education	0.102953
capital.gain	0.097686
hours.per.week	0.085583
x1_Married-civ-spouse	0.064646
x3_Husband	0.048896
capital.loss	0.033387
x1_Never-married	0.028629
x2_Exec-managerial	0.020458

Key point

- Unlike the linear model coefficients, `feature_importances_` **do not have a sign**!
  - They tell us about importance, but not an "up or down".
  - Indeed, increasing a feature may cause the prediction to first go up, and then go down.
  - This cannot happen in linear models, because they are linear.

Do these importances match with importances identified by logistic regression?

In [69]:

```
data = {
    "random forest importance": pipe_rf.named_steps[
        "randomforestclassifier"
    ].feature_importances_,
    "logistic regression importances": pipe_lr.named_steps["logisticreg
    .coef_[0]
    .tolist(),
}
imps = pd.DataFrame(
    data=data,
    index=feature_names,
)
```

In [70]:
```
imps.sort_values(by="random forest importance", ascending=False)[:10]
```

Out[70]:

	random forest importance	logistic regression importances
fnlwgt	0.169580	0.078087
age	0.153339	0.359883
education	0.102953	0.183963
capital.gain	0.097686	2.355403
hours.per.week	0.085583	0.370353
x1_Married-civ-spouse	0.064646	1.324596
x3_Husband	0.048896	-0.032775
capital.loss	0.033387	0.281123
x1_Never-married	0.028629	-0.956018
x2_Exec-managerial	0.020458	1.041384

- Both models agree on `age` , `education` , `capital.gain`
- The actual numbers for random forests and logistic regression are not really comparable.

How can we get feature importances for non `sklearn` models?

- One way to do it is by using a tool called [`eli5` (https://eli5.readthedocs.io/en/latest/overview.html)](https://eli5.readthedocs.io/en/latest/overview.html).

You'll have to install it

```
conda install -c conda-forge eli5
```

Let's look at feature importances for `XGBClassifier` .

In [71]:

```
import eli5

pipe_xgb = make_pipeline(preprocessor, XGBClassifier(random_state=123,
pipe_xgb.fit(X_train, label_encoder.transform(y_train));
eli5.explain_weights(pipe_xgb.named_steps["xgbclassifier"], feature_nam
```

Out[71]:

Weight	Feature
0.4061	x1_Married-civ-spouse
0.0547	capital.gain
0.0441	x3_Own-child
0.0349	education
0.0325	x2_Other-service
0.0268	capital.loss
0.0247	x2_Prof-specialty
0.0179	x2_Exec-managerial
0.0178	x2_Tech-support
0.0172	x2_Handlers-cleaners
0.0164	x2_Machine-op-inspct
0.0164	x2_Farming-fishing
0.0158	x0_Federal-gov
0.0117	age
0.0108	x0_Self-emp-inc
0.0107	hours.per.week
0.0102	x3_Wife
0.0101	sex
0.0094	x3_Not-in-family
0.0091	x0_Self-emp-not-inc
	… 66 more …

Let's look at feature importances for `LGBMClassifier`.

In [72]:

```
pipe_lgbm = make_pipeline(preprocessor, LGBMClassifier(random_state=123
pipe_lgbm.fit(X_train, y_train)
eli5.explain_weights(
    pipe_lgbm.named_steps["lgbmclassifier"], feature_names=feature_name
)
```

Out[72]:

Weight	Feature
0.3558	x1_Married-civ-spouse
0.1910	capital.gain
0.1363	education
0.0852	age
0.0639	capital.loss
0.0418	hours.per.week
0.0245	fnlwgt
0.0134	x2_Exec-managerial
0.0120	x2_Prof-specialty
0.0067	x2_Other-service
0.0065	sex
0.0055	x3_Wife
0.0054	x0_Self-emp-not-inc
0.0052	x2_Farming-fishing
0.0046	x3_Own-child
0.0033	x2_Tech-support
0.0025	x2_Sales
0.0024	x0_Private
0.0024	x0_Federal-gov
0.0023	x2_Handlers-cleaners
	… 66 more …

You can also look at feature importances for `RandomForestClassifier`.

In [73]:
```
eli5.explain_weights(
    pipe_rf.named_steps["randomforestclassifier"], feature_names=featur
)
```

Out[73]:

Weight	Feature
0.1696 ± 0.0113	fnlwgt
0.1533 ± 0.0396	age
0.1030 ± 0.0348	education
0.0977 ± 0.0479	capital.gain
0.0856 ± 0.0250	hours.per.week
0.0646 ± 0.1385	x1_Married-civ-spouse
0.0489 ± 0.1117	x3_Husband
0.0334 ± 0.0157	capital.loss
0.0286 ± 0.0740	x1_Never-married
0.0205 ± 0.0211	x2_Exec-managerial
0.0193 ± 0.0187	x2_Prof-specialty
0.0118 ± 0.0221	sex
0.0110 ± 0.0225	x3_Wife
0.0094 ± 0.0038	x0_Private
0.0093 ± 0.0242	x3_Not-in-family
0.0080 ± 0.0036	x0_Self-emp-not-inc
0.0078 ± 0.0104	x2_Other-service
0.0066 ± 0.0064	x0_Self-emp-inc
0.0066 ± 0.0239	x3_Own-child
0.0064 ± 0.0024	x4_United-States

… 66 more …

Let's compare them with weights what we got with `sklearn` `feature_importances_`

In [74]:
```
data = {
    "Importance": pipe_rf.named_steps["randomforestclassifier"].feature
}
pd.DataFrame(data=data, index=feature_names,).sort_values(
    by="Importance", ascending=False
)[:10]
```

Out[74]:

	Importance
fnlwgt	0.169580
age	0.153339
education	0.102953
capital.gain	0.097686
hours.per.week	0.085583
x1_Married-civ-spouse	0.064646
x3_Husband	0.048896
capital.loss	0.033387
x1_Never-married	0.028629
x2_Exec-managerial	0.020458

- These values tell us globally about which features are important.
- But what if you want to explain a *specific* prediction.
- Some fancier tools can help us do this.

SHAP (SHapley Additive exPlanations)

SHAP (SHapley Additive exPlanations)

- A sophisticated measure of the contribution of each feature.
- Lundberg and Lee, 2017 (https://arxiv.org/pdf/1705.07874.pdf)
- We won't go in details. You may refer to Scott Lundberg's GitHub repo (https://github.com/slundberg/shap) if you are interested to know more.

General idea

Source (https://github.com/slundberg/shap/blob/master/docs/presentations/February%202018%20Talk.ppt:

General idea

- Provides following kind of explanation
  - Start at a base rate (e.g., how often people get their loans rejected).
  - Add one feature at a time and see how it impacts the decision.

Source (https://github.com/slundberg/shap/blob/master/docs/presentations/February%202018%20Talk.ppt:

Let's try it out on tree-based models.

First you'll have to install it.

```
pip install shap
or
conda install -c conda-forge shap
```

Let's create train and test dataframes with our transformed features.

In [75]:
```
X_train_enc = pd.DataFrame(
    data=preprocessor.transform(X_train).toarray(),
    columns=feature_names,
    index=X_train.index,
)
X_train_enc.head()
```

Out[75]:

	age	fnlwgt	capital.gain	capital.loss	hours.per.week	education	sex	x0_Federal-gov
5514	-0.921955	0.632531	-0.147166	-0.21768	-1.258387	8.0	1.0	0.0
19777	-1.069150	-0.186155	-0.147166	-0.21768	-0.447517	8.0	0.0	0.0
10781	-0.185975	-1.085437	-0.147166	-0.21768	-0.042081	13.0	0.0	0.0
32240	-1.216346	-1.575119	-0.147166	-0.21768	-1.663822	12.0	0.0	0.0
9876	-0.553965	1.588701	-0.147166	-0.21768	-0.042081	13.0	1.0	0.0

5 rows × 86 columns

In [76]:
```
X_test_enc = pd.DataFrame(
    data=preprocessor.transform(X_test).toarray(),
    columns=feature_names,
    index=X_test.index,
)

X_test_enc.shape
```

Out[76]: (6513, 86)

Let's get SHAP values for train and test data.

In [77]:
```
import shap

lgbm_explainer = shap.TreeExplainer(pipe_lgbm.named_steps["lgbmclassifi
train_lgbm_shap_values = lgbm_explainer.shap_values(X_train_enc)
```

```
LightGBM binary classifier with TreeExplainer shap values output has chan
ged to a list of ndarray
```

In [78]:
```
train_lgbm_shap_values[1].shape
```

Out[78]: (26048, 86)

In [79]:
```
test_lgbm_shap_values = lgbm_explainer.shap_values(X_test_enc)
test_lgbm_shap_values[1].shape
```

```
LightGBM binary classifier with TreeExplainer shap values output has chan
ged to a list of ndarray
```

Out[79]: (6513, 86)

- For classification it's a bit confusing. It gives SHAP arrays both classes.
- Let's stick to shap values for class 1, i.e., income > 50K.

For each example and each feature we have a SHAP value.

In [80]:
```
train_lgbm_shap_values[1]
```

Out[80]:
```
array([[-4.23243013e-01, -5.89878323e-02, -2.65263112e-01, ...,
         9.63030623e-04,  0.00000000e+00,  5.74466631e-04],
       [-6.83190014e-01,  1.15708200e-02, -2.72482485e-01, ...,
         8.17274476e-04,  0.00000000e+00,  8.09406158e-04],
       [ 4.49106369e-01, -1.32455245e-01, -2.39454581e-01, ...,
         8.27603313e-04,  0.00000000e+00,  4.22023416e-03],
       ...,
       [ 1.02714900e+00,  2.38119557e-02, -1.88163464e-01, ...,
         1.13580827e-03,  0.00000000e+00,  6.94390861e-04],
       [ 6.37084418e-01,  2.90573592e-02, -3.03429292e-01, ...,
         9.70726909e-04,  0.00000000e+00,  2.16856964e-03],
       [-1.24950883e+00,  1.19867799e-01, -2.23378846e-01, ...,
         9.70674774e-04,  0.00000000e+00,  9.73838044e-04]])
```

Let's look at the average SHAP values associated with each feature.

In [81]:
```
values = np.abs(train_lgbm_shap_values[1]).mean(0)
pd.DataFrame(data=values, index=feature_names, columns=["SHAP"]).sort_v
    by="SHAP", ascending=False
)[:10]
```

Out[81]:

	SHAP
x1_Married-civ-spouse	1.086269
age	0.823933
capital.gain	0.572778
education	0.409543
hours.per.week	0.313901
sex	0.188874
capital.loss	0.138607
x3_Own-child	0.112871
x2_Exec-managerial	0.107399
x2_Prof-specialty	0.098181

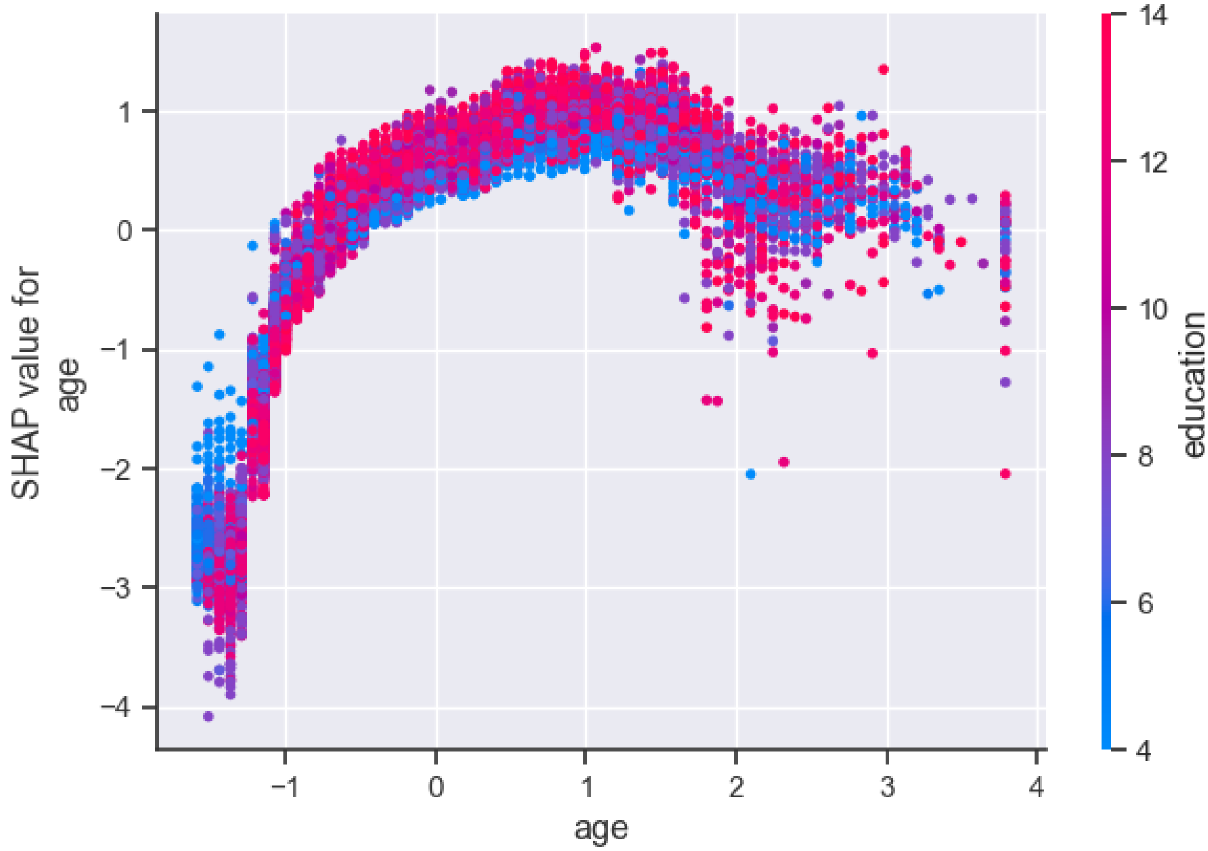
You can think of this as global feature importances.

SHAP plots

In [82]:
```
# load JS visualization code to notebook
shap.initjs()
```

Dependence plot

In [83]:
```
shap.dependence_plot("age", train_lgbm_shap_values[1], X_train_enc)
```



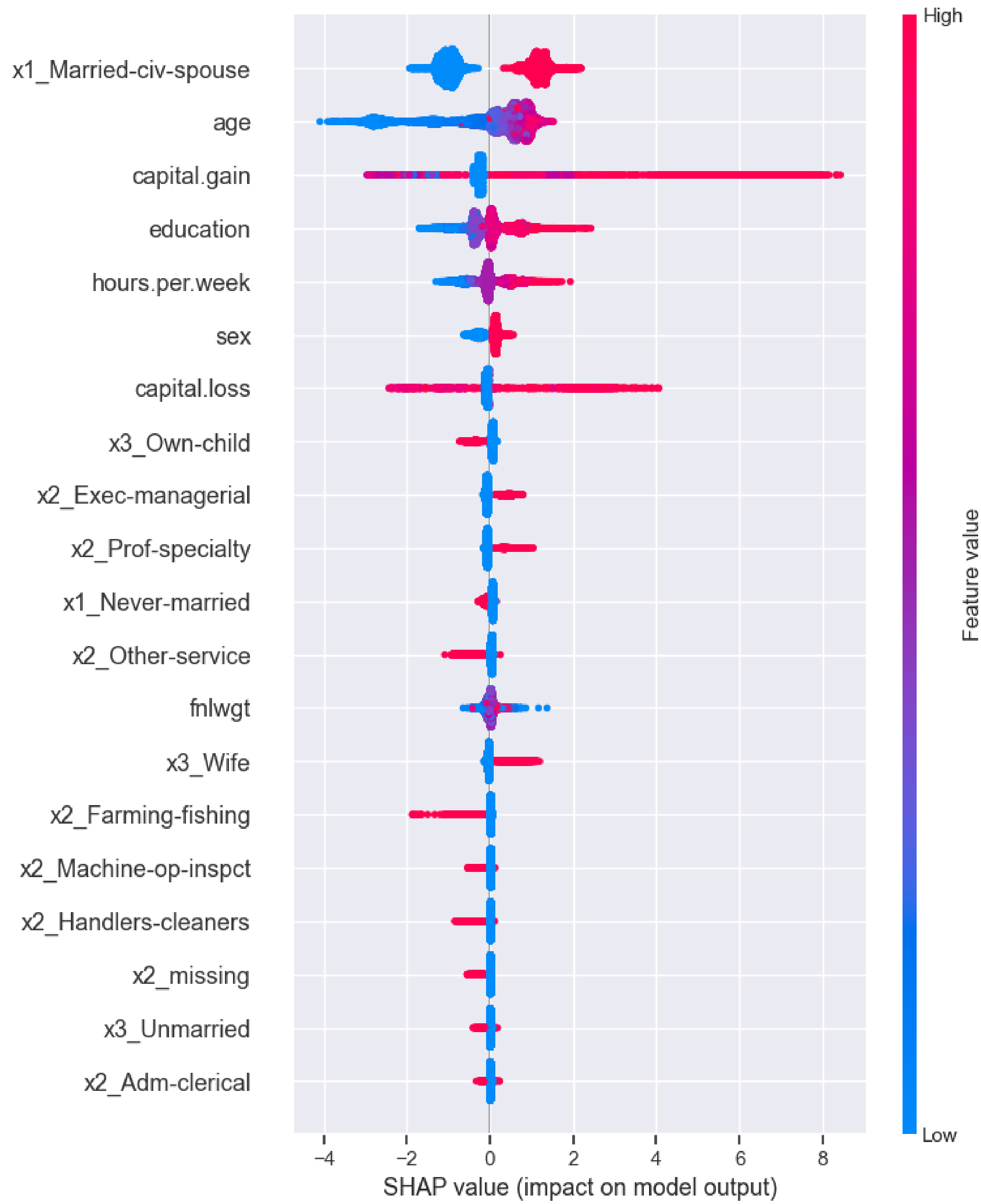
The plot above shows effect of `age` feature on the prediction.

- Each dot is a single prediction for one example.
- The x-axis represents values of the feature age (scaled).
- The y-axis is the SHAP value for that feature, which represents how much knowing that feature's value changes the output of the model for that example's prediction.
- Lower values of age have smaller SHAP values for class ">50K".
- Similarly, higher values of age also have a bit smaller SHAP values for class ">50K", which makes sense.
- There is some optimal value of age between scaled age of 1 which gives highest SHAP values for for class ">50K".

Summary plot

In [84]:
```
shap.summary_plot(train_lgbm_shap_values[1], X_train_enc)
```

```
No data for colormapping provided via 'c'. Parameters 'vmin', 'vmax' will
be ignored
```



The plot shows the most important features for predicting the class. It also shows the direction of how it's going to drive the prediction.

- Presence of the marital status of Married-civ-spouse tends to have a large positive SHAP value (makes class 1 more likely), and absence of marital status seems to have negative SHAP values for class 1.
- Higher levels of education seem to have larger SHAP values for class 1, whereas smaller levels of education have smaller SHAP values.

Force plot

- Let's try to explain predictions on a couple of examples from the test data.
- I'm sampling some examples where the target is <=50K and some examples where the target is >50K.

In [85]:
```
y_test_reset = y_test.reset_index(drop=True)
y_test_reset
```

Out[85]:
```
0       <=50K
1       <=50K
2       <=50K
3       <=50K
4       <=50K
        ...
6508    <=50K
6509    <=50K
6510     >50K
6511    <=50K
6512     >50K
Name: income, Length: 6513, dtype: object
```

In [86]:
```
l50k_ind = y_test_reset[y_test_reset == "<=50K"].index.tolist()
g50k_ind = y_test_reset[y_test_reset == ">50K"].index.tolist()

ex_l50k_index = l50k_ind[10]
ex_g50k_index = g50k_ind[10]
```

Example with prediction <=50K

In [87]:
```
pipe_lgbm.named_steps["lgbmclassifier"].predict_proba(X_test_enc)[ex_l5
```

Out[87]:
```
array([0.98998011, 0.01001989])
```

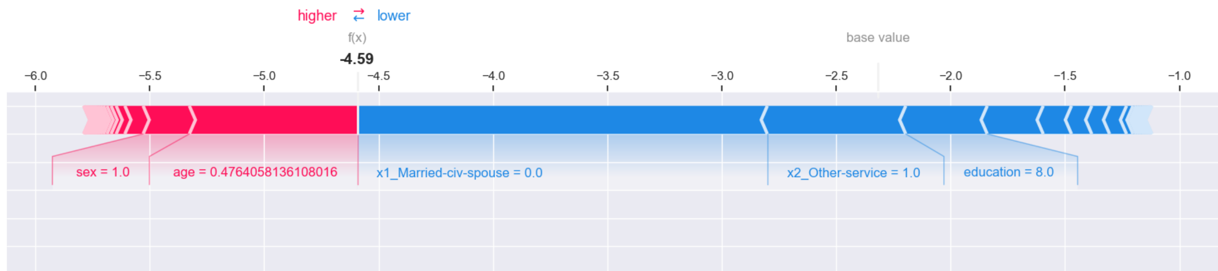
In [88]:
```
pipe_lgbm.named_steps["lgbmclassifier"].predict(X_test_enc, raw_score=T
    ex_l50k_index
]  # raw score of the model
```

Out[88]:
```
-4.59311253704159
```

In [89]:
```
lgbm_explainer.expected_value[1]
```

Out[89]: -2.3163172510079377

In [90]:
```
shap.force_plot(
    lgbm_explainer.expected_value[1],
    test_lgbm_shap_values[1][ex_l50k_index, :],
    X_test_enc.iloc[ex_l50k_index, :],
    matplotlib=True,
)
```



Example with prediction >50K

In [91]:
```
pipe_lgbm.named_steps["lgbmclassifier"].predict_proba(X_test_enc)[ex_g5
```

Out[91]: array([0.47228832, 0.52771168])

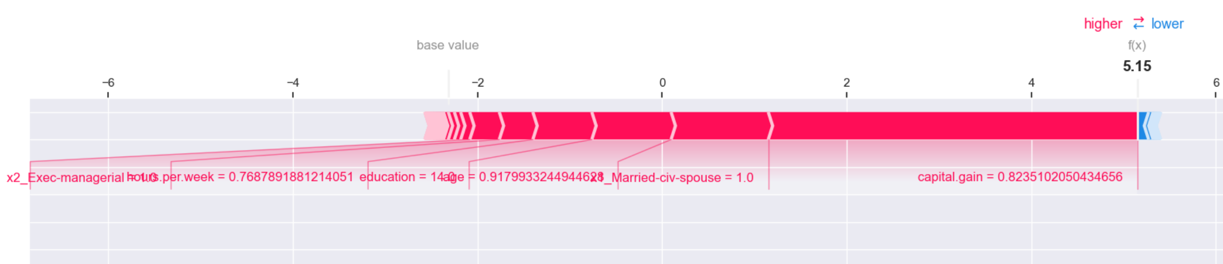
In [92]:
```
pipe_lgbm.named_steps["lgbmclassifier"].predict(X_test_enc, raw_score=T
    ex_g50k_index
]  # raw model score
```

Out[92]: 0.11096043410156158

In [93]:
```
g50k_ind[:10]
```

Out[93]: [17, 18, 30, 31, 39, 45, 49, 58, 59, 62]

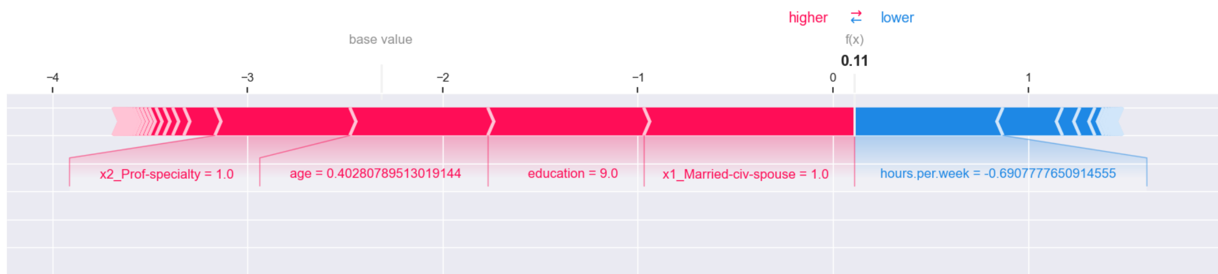
In [94]:
```
shap.force_plot(
    lgbm_explainer.expected_value[1],
    test_lgbm_shap_values[1][18, :],
    X_test_enc.iloc[18, :],
    matplotlib=True,
)
```



In [95]:
```
test_lgbm_shap_values[1][ex_g50k_index, :],
```

Out[95]:
```
(array([ 7.06576926e-01,  4.48050353e-02, -3.04826888e-01, -8.90345564e-0
2,
        -7.60234999e-01,  7.99009432e-01,  1.59145115e-01, -1.04561236e-0
2,
         1.12544519e-02,  0.00000000e+00, -7.93711497e-03, -8.36423421e-0
3,
        -4.36341788e-02,  7.59578850e-03,  0.00000000e+00,  2.58025291e-0
3,
        -1.21233828e-03,  0.00000000e+00,  1.08073198e+00, -1.96300214e-0
3,
         4.05063596e-02,  3.89528728e-03, -3.40493462e-03,  1.13966166e-0
2,
         0.00000000e+00,  1.78540517e-02, -9.45890343e-03,  2.07052478e-0
2,
         4.68861970e-03,  2.04262116e-02,  5.73720200e-02,  4.04716758e-0
3,
         6.91720232e-01, -4.07221602e-03, -1.66621103e-02, -1.42348633e-0
2,
         8.08173720e-03,  1.52914770e-02,  8.16100692e-03, -1.52031143e-0
2,
         4.30523314e-03,  3.49488337e-02,  1.95922173e-02, -7.73110056e-0
2,
         0.00000000e+00, -1.86454897e-04,  9.32582520e-04,  2.43918385e-0
3,
        -3.25037985e-04,  1.48300302e-03,  0.00000000e+00,  0.00000000e+0
0,
         0.00000000e+00,  0.00000000e+00, -2.05106608e-04,  2.38263534e-0
4,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+0
0,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+0
0,
         0.00000000e+00, -9.45855608e-04,  0.00000000e+00,  0.00000000e+0
0,
         0.00000000e+00,  7.16936676e-03,  0.00000000e+00,  0.00000000e+0
0,
         0.00000000e+00, -2.83890851e-03,  0.00000000e+00,  0.00000000e+0
0,
         2.26468523e-03,  0.00000000e+00,  2.02526698e-04,  0.00000000e+0
0,
         0.00000000e+00,  0.00000000e+00,  9.58695151e-03,  9.66566029e-0
4,
         0.00000000e+00, -1.84802303e-04]),)
```

In [96]:
```
shap.force_plot(
    lgbm_explainer.expected_value[1],
    test_lgbm_shap_values[1][ex_g50k_index, :],
    X_test_enc.iloc[ex_g50k_index, :],
    matplotlib=True,
)
```



Observations:

- Everything is with respect to class 1 here.
- The base value for class 1 is -2.316. (You can think of this as the average raw score.)
- We see the forces that drive the prediction.
- That is, we can see the main factors pushing it from the base value (average over the dataset) to this particular prediction.
- Features that push the prediction to a higher value are shown in red.
- Features that push the prediction to a lower value are shown in blue.

Note: a nice thing about SHAP values is that the feature importances sum to the prediction:

In [97]:
```
test_lgbm_shap_values[1][ex_g50k_index, :].sum() + lgbm_explainer.expec
```

Out[97]: 0.11096043410156309

Provides explainer for different kinds of models

- [TreeExplainer (https://shap.readthedocs.io/en/latest/)](https://shap.readthedocs.io/en/latest/) (supports XGBoost, CatBoost, LightGBM)
- [DeepExplainer (https://shap.readthedocs.io/en/latest/index.html#shap.DeepExplainer)](https://shap.readthedocs.io/en/latest/index.html#shap.DeepExplainer) (supports deep-learning models)
- [KernelExplainer (https://shap.readthedocs.io/en/latest/index.html#shap.KernelExplainer)](https://shap.readthedocs.io/en/latest/index.html#shap.KernelExplainer) (supports kernel-based models)
- [GradientExplainer (https://shap.readthedocs.io/en/latest/index.html#shap.GradientExplainer)](https://shap.readthedocs.io/en/latest/index.html#shap.GradientExplainer) (supports Keras and Tensorflow models)

- Can also be used to explain text classification and image classification
- Example: In the picture below, red pixels represent positive SHAP values that increase the probability of the class, while blue pixels represent negative SHAP values the reduce the probability of the class.

Source (https://github.com/slundberg/shap)

Other tools

- lime (https://github.com/marcotcr/lime) is another package.

If you're not already impressed, keep in mind:

- So far we've only used sklearn models.
- Most sklearn models have some built-in measure of feature importances.
- On many tasks we need to move beyond sklearn, e.g. LightGBM, deep learning.
- These tools work on other models as well, which makes them extremely useful.

Why do we want this information?

Possible reasons:

- Identify features that are not useful and maybe remove them.
- Get guidance on what new data to collect.
    - New features related to useful features -> better results.
    - Don't bother collecting useless features -> save resources.
- Help explain why the model is making certain predictions.
    - Debugging, if the model is behaving strangely.
    - Regulatory requirements.
    - Fairness / bias.
    - Keep in mind this can be used on **deployment** predictions!

? ? Questions for you

True/False

1. You train a random forest on a binary classification problem with two classes [neg, pos]. A value of 0.580 for feat1 given by `feature_importances_` attribute of your model means that increasing the value of feat1 will drive us towards positive class.
2. eli5 can be used to get feature importances for non `sklearn` models.
3. With SHAP you can only explain predictions on the training examples.

In [ ]:
```
 
```